UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1397

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes

A DATABASE RULE LANGUAGE COMPILER SUPPORTING PARALLELISM

Jean-Pierre CHEINEY
Gerald KIERNAN
Christophe de MAINDREVILLE

Février 1991

RDL//C : Un Langage de Programmation Parallèle pour SGBD Relationnels

Jean-Pierre Cheiney[*], *Gerald Kiernan*[**], *Christophe de Maindreville*[**]

Résumé : Ce rapport présente l'extension apportée à un compilateur de règles pour supporter le parallélisme. Le langage de règles est appelé RDL//C. Il est basé sur une version étendue du calcul relationnel de tuples. Le granule de parallélisme introduit est la règle de production. L'utilisateur spécifie dans un langage de contrôle, le parallélisme qu'il désire apporter à son programme de règles. De son point de vue, le parallélisme consiste en la définition d'un ordre partiel sur l'exécution des règles. Le compilateur traduit les programmes de règles en programmes C/SQL qui s'exécutent au dessus de SGBD relationnels. Notre implémentation a été effectué au sein d'un réseau de stations SUN qui correspond à une architecture shared nothing MIMD. Le programme C généré par le compilateur se connecte aux SGBD sur les différentes stations SUN. Les performances initiales du système sont présentées. Elles montrent les avantages et limitations d'une telle approche. La contribution de cette recherche est de montrer comment le parallélisme peut être supporté au sein d'un programme de règles pour SGBD dans un environnement distribué.

Mots clés : Règles de production, parallélisme, langage de contrôle, compilateur, SGBD.

A Database Rule Language Compiler Supporting Parallelism

Jean-Pierre Cheiney, Gerald Kiernan, Christophe de Maindreville

Abstract : *This paper presents an extension to a database rule language compiler to support parallelism. The rule language is called RDL//C and is based on an extended version of tuple relational calculus. The grain of parallelism introduced into RDL//C is the production rule. The user is made aware of parallel processing by having to specify in a sub-language of control which sets of rules are to run in parallel and in which order. From the user view point, the parallelism is managed as a partial order over a set of rules. The compiler translates rule programs into C based applications which run over the DBMS. Our implementation is over a set of UNIX[†] workstations which corresponds to a MIMD shared-nothing architecture. Parallelism is achieved by having the application connect to different DBMS servers residing on workstations on a local network. Initial performance results are presented. They show the advantages and limitations of parallel execution in this architecture. The main contributions of the paper are (i) the definition of a sub-language of control to specify a parallel execution of a rule program and (ii) to show how parallel extensions can be brought to a rule language compiler using a standard distributed environment with a relational DBMS.*

Key words : *Production Rules, Parallelism, Language of Control, Compilation, RDBMS, Implementation.*

---

[*] Ecole Nationale Supérieure des Télécommunications - 46 rue Barrault -75013 Paris - FRANCE and INRIA Sabre Project

[**] Institut National de la Recherche en Informatique et Automatique - INRIA Sabre Project - Rocquencourt , 78153 Le Chesnay Cedex - FRANCE.

[†] UNIX is a trademark of AT&T Bell Laboratories

A Database Rule Language Compiler Supporting Parallelism

J.-P. Cheiney[*], *G. Kiernan*[**], *C. de Maindreville*[**]

Abstract : *This paper presents an extension to a database rule language compiler to support parallelism. The rule language is called RDL//C and is based on an extended version of tuple relational calculus. The grain of parallelism introduced into RDL//C is the production rule. The user is made aware of parallel processing by having to specify in a sub-language of control which sets of rules are to run in parallel and in which order. From the user view point, the parallelism is managed as a partial order over a set of rules. The compiler translates rule programs into C based applications which run over the DBMS. Our implementation is over a set of UNIX*[†] *workstations which corresponds to a MIMD shared-nothing architecture. Parallelism is achieved by having the application connect to different DBMS servers residing on workstations on a local network. Initial performance results are presented. They show the advantages and limitations of parallel execution in this architecture. The main contributions of the paper are (i) the definition of a sub-language of control to specify a parallel execution of a rule program and (ii) to show how parallel extensions can be brought to a rule language compiler using a standard distributed environment with a relational DBMS.*

Key words : *Production Rules, Parallelism, Language of Control, Compilation, RDBMS, Implementation.*

1. Introduction

Rule languages have long been recognized as candidates for parallel languages [Almasi89, Ganguly90, Wolfson90]. Parallel evaluation of Datalog programs has been studied in [Ganguly90, Wolfson90]. These papers introduce the notion of *discriminating predicate* which allows the instanciations of rules to be partitioned among different processors. Both papers are concerned with the reduction of communication overhead and load balancing between the processors and discuss the trade-off between non-redundancy and communication costs. One important result stated in [Wolfson90] is the undecidability of the decomposition problem. The parallel evaluation of Datalog-neg programs is discussed in [Wolfson90]. For such programs, in general, the processors have to be synchronized at each stratum. This means that each processor has to wait until all the processors have completed their computation before proceeding to the next stratum.

* Ecole Nationale Supérieure des Télécommunications - 46 rue Barrault -75013 Paris - FRANCE and INRIA Sabre Project

** Institut National de la Recherche en Informatique et Automatique - INRIA Sabre Project - Rocquencourt , 78153 Le Chesnay Cedex - FRANCE.

† UNIX is a trademark of AT&T Bell Laboratories

Experiments with parallelism in production systems such as OPS5 [Brownston85] have been presented in [Gupta89]. In such environments, it is up to the system to detect the possible sources of parallelism in a rule program. It is also the system's job to guarantee that the parallel execution of a rule program is equivalent to the sequential one. Different sources of parallelism have been identified [Almasi89]. The most obvious one is the production rule as the unit of parallelism. Production rules which can run independently are detected and each group is assigned a processor. Three other sources of parallelism are AND-parallelism, OR-parallelism, and Parallel pattern matching. These last three sources of parallelism concern intra-rule parallelism. A single production rule is decomposed into sub-parts which can run independently. The first two types of parallelism come from AND/OR graphs which represent the connection between the predicates and rules which comprise a production system. In the case of Horn clauses, for example, the ANDs are the conjunction of expressions which form the body of a rule; and the ORs are rules with identical heads. Parallel pattern matching is the implementation of a parallel version of the inference engine.

In [Almasi89], the authors report the claim that a production rule application which is written with parallel processing in mind yields a higher degree of useful parallelism than one which is written with a serial machine in mind. Moreover, the performance gained is proportional to the number of rules in the system; and of course to the number of processing elements. They also claim that the useful parallelism and the speed-up from the "rule-per-processor-approach" is yet to be fully established. This indicates that while it is up to the system to detect and implement parallelism, the user must know how the system achieves this parallelism in order to exploit it fully. In [Srivastava89], two types of parallelism are proposed for database production systems. These are user-visible and user-transparent categories. It is the user's responsibility to divide a task into non-interacting subtasks and it is the system's responsibility to execute each element of a subtask in parallel.

The approach presented in this paper supports parallelism at the rule level. The user is made aware of parallel processing by having to specify which sets of rules are to run in parallel and in which order. A control sub-language is used for this purpose. We propose that a parallel algorithm is better than a sequential one on which a certain degree of parallelism might be automatically extracted. We do not address in this paper the automatic detection of parallelism in rule programs.

This paper presents an extension to a rule language compiler to support parallelism. A prototype of the RDL//C compiler is operational on a network of UNIX workstations [Unix84]. Since RDL/C [Kiernan90b, Kiernan91] production rules are based on relational calculus, they can be solved by a relational DBMS without having to extract data during the inference cycle. The inference is driven from the application (generated by the compiler) through a series of calls to the DBMS using an SQL interface. The compiler manages parallelism in production rules by generating a program which connects to several DBMS over a network of workstations. In this framework, each DBMS is a parallel processing unit. The application communicates with each DBMS using standard UNIX interprocess communication facilities called sockets. To run a set of rules in parallel, the application

process forks (fork is a UNIX system call which creates a new process from a current process) a number of times equal to the number of sets of rules which are to be run in parallel. The system architecture is an MIMD share-nothing architecture. Intra-rule parallelism cannot be supported without modification of the underlying DBMS, it is therefore not discussed in this paper.

The main contribution of the paper is to show how a parallel production system can be implemented over a network of workstations. Initial performance results indicate the advantages of this approach for high performance production rule systems. Section 2 presents an overview of the production rule language which is used throughout this paper. Production Compilation Networks (PCN) [Maindreville88] as a model of parallelism are introduced in section 3. PCN are the computational model for the RDL//C production rule system. Section 4 describes the syntactic extensions that were brought to the serial version of the compiler to support parallelism. Section 5 concerns the architecture. The compiler and the run-time support are presented. Section 6 details the implementation and section 7 gives performance results and discuss the advantages and limitations of this approach.

2. Overview of the Language

RDL//C is derived from RDL1 [Maindreville88, Kiernan90a], a production rule language which has been integrated in the Sabrina RDBMS [Gardarin89]. The RDL//C language supports declarative programming based on RDL1 and procedural programming based on C code. In this section, we present an overview of the language through some examples. The syntax and semantics of RDL//C are given in [Kiernan90b, Kiernan91].

2.1. Illustrative Example

Consider the base relation Parent having the schema Parent (asc integer, desc integer). The following rule program computes the transitive closure of the Parent relation :

```
MODULE ancestor ;
BASE
        Parent (asc int, desc int);
DEDUCED
        Ancestor LIKE Parent;

RULES
r1 is
    IF Parent (x)
        THEN + Ancestor (x) ;
r2 is
    IF Parent (x) and Ancestor (y) (x.desc = y.asc)
        THEN + Ancestor (asc = x.asc, desc = y.desc);
END MODULE
```

A parallel version of this program will be presented in section 7, when performance issues are discussed.

2.2. The Kernel Language

2.2.1. The Syntax

The rule part of an RDL//C program is composed of a set of if-then rules. The IF part of a rule is a tuple relational calculus expression. Its syntax is very close to the syntax of a WHERE clause in the SQL language. The THEN part of a rule is a set of *actions* that are insertions, deletions of tuples in relations, procedural side-effects and variable assignments. A discussion of the latter two types of actions are beyond the scope of this paper. However, they are described in [Kiernan90b]. The action part is very close to the SELECT clause of an SQL statement.

Let Person and Worker, be two relations having the same schema (id integer, name char, age integer).

The following expressions are valid Left-Hand-Sides (LHS) of rules :

```
Person(x) and Worker(y) (x.id = y.id)
Foreach x in Person (x.age + 1 > 20)
Person(x) (x.age > 20)
```

The Right-Hand Side (RHS) of a production rule supports two elementary actions, denoted "+" and "-". The update action "+" takes a set of facts and maps a database state into another state which contains these facts. On the contrary, the action "-" takes a set of facts and deletes it from a relation. A *multiple action* consists in a sequence of actions.

The following expressions are valid RHS of rules :

```
+Person(x)
-Person(x)  +Person(id = x.number, name = x.name, age = x.age + 1)
+Person(x) + Person(y) - Worker(x)
```

Following is a set of valid rules :

```
If Person(x) then + Worker(x) ;
If Person(x) ( x.id = 4) then - Worker(x) ;
If Person(x) then + Person(x)  -Person(id= x.id, name= x.name, age= x.age +1);
```

2.2.2. The Semantics

The semantics adopted for the RDL//C language is a set oriented one: When a condition is evaluated against the database, it returns the set of instances which make the condition valid. When an action is executed against the database, it is executed for all the values which appear as arguments in the action. In the following, we present examples of rule execution.

Let us consider the following rules :

Firing the following rule causes the insertion into the Worker relation of the contents of the Person relation .

```
if Person(x) then + Worker(x) ;
```

Firing the following rule causes the deletion from the R1 relation of the contents of the Person relation.

```
if Person(x)  then - R1(x) ;
```

Firing the following rule leads to a null action.

```
if Person(x)  then - R1(x)  + R1(x) ;
```

Firing the following rule

```
if Person(x) and R1(y) and x.att1 = 10 and x.att2 = y.att2
then +Q1(att1 = x.att1, att2 = y.att1) - R1(y) ;
```

causes the insertion into the Q1 relation the set of tuples :

```
I = {x.att1, y.att2/Person(x) and R1(y) and x.att1 = 10 and x.att2 = y.att2 }
```

and the deletion from R1 the set

```
D = {x | Person(x) and R1(y) and x.att1 = 10 and x.att2 = y.att2}
```

2.3. Partial Ordering of Rules

The mixing of declarative reasoning and imperative control has been shown necessary for many applications. A control language has been designed to specify an application mode over the rules. It induces a partial ordering over the rules. Each expression of the sub-language is declared in the module. The basic terms of control language are rule names. A general expression exp in the control language is :

```
block (exp) means that exp has to be fired until a fixpoint is reached.
seq (exp1,exp2 ) means that exp1 is fired once and then exp2 is fired once.
```

If a rule name does not appear in the control part of the program, its firing is chosen at random by the inference engine. If there is no CONTROL section, the rule interpreter applies a default strategy. The priority of rules is according to the order of their appearance in the module.

Let us consider a rule program {r1, r2, r3, r4, r5}.

seq (r1, block (r2, r3, r4), r5) is a possible expression. It enforces a computation of the form : $(r1)^a ((r2)^*(r3)^*(r4)^*)^\sigma (r5)^a$ using standard notation for context-free grammars. The notation $()^\sigma$ stands for "fire up to saturation", * is equal to N or 0 and a is equal to 1 or 0.

3. The PCN Model of Parallelism

The PCN is used as an execution model for the RDL//C language. It provides the inference engine with a graphical representation of a rule program. The PCN model has been introduced in [Maindreville88]. The PCN model is a Petri Net based model and derived from Predicate Transition Nets (PrTN). A formal definition of PrTN can be found in [Genrich86]. The structure of a PCN represents the relationships between rules and relational predicates which occur in a rule program. We represent each rule by a transition and each relational predicate involved in the rule by a place. The relational predicates that occur in the condition part of a rule are input places to the transition representing the rule and the relational predicates that occur in the action part of the rule are the output places of the transition. The condition of a rule is represented in the transition's condition. Figure 3.1 represents the PCN associated to the rule program which computes the transitive closure of the Parent relation. The initializing rule is modelled by the transition T1. The recursive rule is modelled by the transition T2. Place P corresponds to the Parent relation and Place A corresponds to the Ancestor relation.

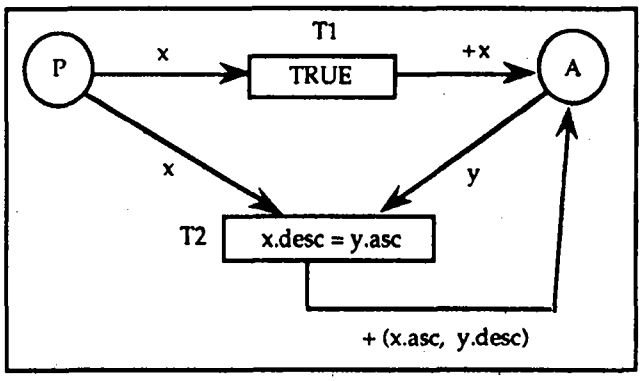

Figure 3.1: *PCN for the Ancestor rules*

A *marking* is a distribution of tokens over the places of a PCN. The token is a basic concept in the PCN model, and represents a database tuple. Traditionally, a Petri Net based model proceeds by firing transitions. A transition can be fired if it is enabled. A transition firing produces tokens according to the labels of the output edges. For a given initial marking, a place P has a *stable marking* iff none of its input transitions is enabled. A transition T is said to be *stable* for a given initial marking iff all its output places have a stable marking. When the PCN reaches a stable marking, no transition can be enabled and the PCN evaluator procedure halts. This corresponds to a fixpoint for the set of rules modelled by the net.

The control language defined in the previous section can be mapped onto a PCN [Maindreville88]. Rule names are then replaced by transition names. A sequence of the control language is called annotation. A PCN and its annotation constitutes an *annotated PCN*.

The PCN model supports parallelism at the rule level as it is shown in the following example: consider the PCN,

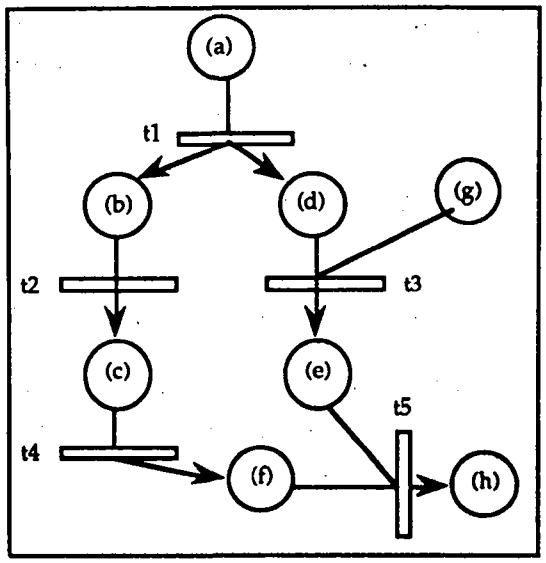

Figure 3.2 : *PCN*

The firing of the transitions t2 and t3 is easily parallelized since they do not share any post places in the PCN structure. On the contrary, the transitions t4 and t5 are not easily parallelized since the firing of t5 needs the contents of the place named f which is given by the firing of t4. Such a restriction is due to the set oriented computation of each transition (rule). A dataflow execution would have allowed the relaxation of this kind of limitation.

We now give syntactic restrictions over the PCN structure that allow the parallelization of a rule program. The limitations are of two different kinds. The first one is due to the parallel granularity we chose, i.e. the rule. The second one is for semantic considerations. As discussed in [Wolfson90], for non deterministic languages such as Datalog-neg, the processors have to be synchronized at each stratum. As RDL//C is an extension of Datalog-neg, the same limitation applies : a non recursive transition is fired iff its input places have reached a stable marking. As shown in [Maindreville88], this induces a partial ordering on the rule program and, for instance, implements the notion of stratification. This limitation synchronizes read/write operations between rules. That is, a rule is not allowed to write data in a relation which is read by another rule in parallel (see rules t4 and t5 in the previous section).

These limitations are expressed over the PCN model as follows:

Let us consider {ti} a set of transitions, and Pre(ti) and Post(ti) the input and output places of ti. Then tl and tk can be parallelized (noted by tl // tk), iff:

- $Post(tl) \supset Pre(tk)$ or $Post(tk) \supset Pre(tl)$
- $Post(tl) \neq Post(tk)$ for each l, k

These two conditions allow only independent rules to be fired in parallel. They ensure that the semantics of the rule program are preserved by the parallel processing. On the PCN displayed in figure 3.2, only transitions (rules) t2, t4 and t3 can be run in parallel.

4. Extending the Language to Support Parallelism

The RDL/C language has been extended to support inter rules parallelism. The parallelism is specified in the language of control as an explicit partial ordering over the set of rules.

Before running a rule module in parallel, the user must know how many, and on which machines the module is to run. This is given in the ON statement after the module declaration. The absence of this statement indicates to the compiler that the module will not run in parallel. The following is an example of the ON statement:

```
module ancestor;
on servA, servB;
```

This declaration states that the module can run in parallel on two servers called servA and servB. The language extension used to specify parallelism among the rules is the PAR structure, found in the control string. PAR accepts up to N arguments (N = the number of servers in the ON statement). Each argument is itself a control argument. PAR cannot be nested. Semantic checks are done over the arguments to ensure that the rules to be run in parallel are independent as defined in the previous section. Consider the following example:

```
SEQ (r0, PAR (SEQ (r1, BLOCK (r2)), SEQ (r1B, BLOCK (r2B))), r3)
```

This control structure specifies two sets of rules to be run in parallel. The first set contains rules r1 and r2; the second set contains rules r1B and r2B. The control structure of the first set states that rule r1 fires once and that rule r2 fires up to saturation. Rules r1 and r2 will run on the server servA and rules r1B and r2B will run on the server servB. When a set of rules has been fired according to the control structure given in the control string, the process controlling it attempts to synchronize with other processes running other sets in parallel. All rules in the PAR control structure must have fired according to the control string before the engine attempts to fire rule r3. As stated earlier, PAR cannot be nested but can, however, appear more than once in the control string. Consider the following control string.

```
SEQ (r0, PAR (r1, r2), r3, PAR (r4, r5), r6)
```

In the above example, the engine will attempt to fire rule r0, and then fire rules r1 and r2 in parallel. When r1 and r2 are no longer firable, the engine will move on to rule r3 and then, attempt to fire rules r4 and r5 in parallel before firing r6. The following control structure is equivalent to the previous one.

```
SEQ (r0, PAR (BLOCK (r1), BLOCK (r2)), r3, PAR (BLOCK (r4), BLOCK (r5)), r6)
```

A rule which is run in parallel can also be run sequentially in the same module. This is the case for rule r2 in the following example.

```
SEQ (r0, r1, PAR (r2, r3), r4, r2, r5)
```

The inference engine will attempt to fire rules r2 and r3 in parallel. It will then attempt to fire rule r4 before attempting to fire rule r2 again.

5. Architecture

5.1. Overview of the Compiler

The general architecture of the RDL//C compiler is portrayed in Figure 5.1.

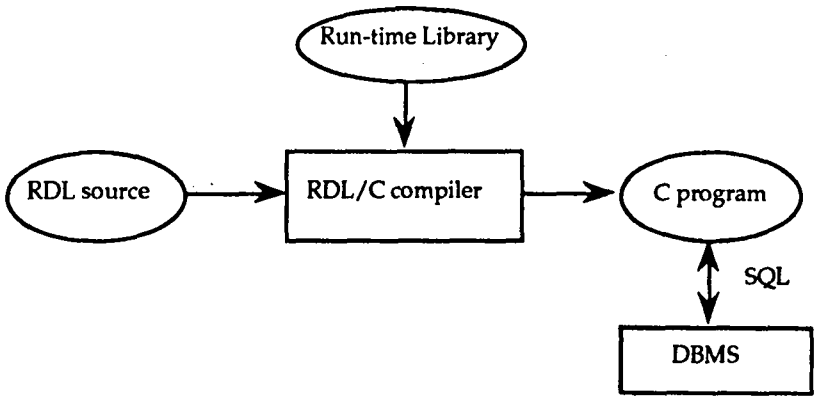

Figure 5.1 : *Sketch of the RDL//C compilation environment.*

The compiler accepts a source program and produces, as output, a C program which implements the rule program. The C program contains code to implement each rule and includes the inference engine which fires rules until a fixpoint is reached. The DBMS does not require any inferencing capabilities to process the program. The extraction of the data from the DBMS to the application is not required during the inference process. This is because rules are based on relational calculus and can thus be solved by the DBMS.

The compiler generates an SQL connection statement for each server specified in the ON statement in the module. The PAR structure is compiled from the control string in basically the same way as are the other two structures SEQ and BLOCK. The run-time library which is linked with the C program produced by the compiler manipulates parallelism. These procedures are the topic of the next section.

5.2. Run-time Environment

The run-time environment is a set of workstations linked to one another through a network. Each workstation has similar computational power. The application resides on one workstation while each workstation is assigned one DBMS process. Each DBMS has its local database. Each relation referenced in a module must be found in at least one site. This allows a horizontal and vertical partitioning of relations in so long as each partition is a named relation existing on a site. The

DBMS is a standard relational DBMS. Although the one that we are using supports Abstract Data Types [Gardarin89], this is not relevant for parallel processing. Parallelism is achieved by having the application access each DBMS concurrently.

5.2.1. UNIX System Tools

This architecture is supported by the UNIX operating system. The key facilities used to support parallelism are the process control facilities and network facilities. The UNIX *fork* statement creates a duplicate process from a running process. Running concurrent processes communicate using sockets and pipes. Sockets and pipes are similar structures used to support interprocess communication. The Network File System (NFS) allows users on different machines to share files.

5.2.2. Schema of the Run-Time Environment

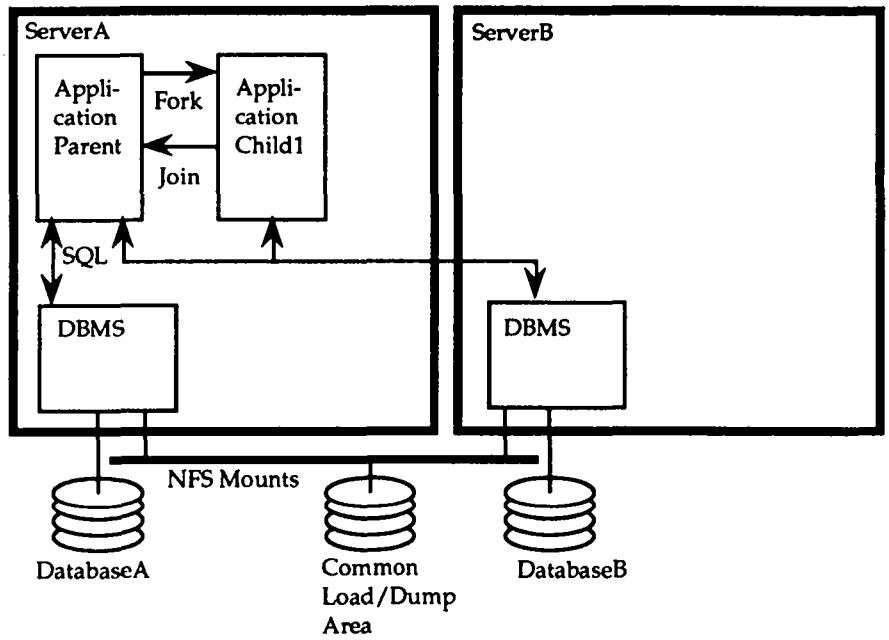

Figure 5.2 : *The Run-Time Environment.*

5.2.3. The Application and the DBMSs

An application requiring access to the database issues an SQL *connect* statement to establish a connection to the DBMS. The application and the DBMS then communicate through sockets. The application issues SQL commands to the DBMS and the DBMS returns its results after each command. There is normally one such connection between an application and the DBMS. However, to exploit the power of parallel processing, the compiler generates an application which

establishes a connection with DBMS residing on different machines. The application can thus communicate with each DBMS by using the communication link that is assigned to it.

5.2.4. The Parent Application and its Children

Although the application can access each DBMS independently, it cannot do so in parallel without creating extra processes. To run a set of rules in parallel, the parent (or main) process creates a child process for each set of rules which are to be run in parallel. This is achieved with the UNIX fork system call. Each child is assigned a particular DBMS. All application processes reside on the same machine regardless of the fact that they access DBMS which are on different machines. Before creating a child, the parent dumps all the relations which are referenced in the set of rules managed by the child. The child loads the relations before starting to process the rules. Running parallel processes synchronize when they have finished running the set of rules they have been assigned. Each child must return information on the rules that it fired; this communication is done through pipes. One such pipe is created per child process. Relations which were created by a child process must be made available to the parent. This is achieved by having the child dump these relations on a common file system and the parent load them from this same file system. The restrictions over the PCN avoid any collisions between relations being dumped. Two rules running in parallel can not reference a same relation in the RHS. After having done this, the child process exits. The parent process synchronizes with the child processes once it has finished running the set of rules that were assigned to it. The parent synchronizes with each child, one after the other. Once it has finished, the parent process can resume firing rules in sequence or in parallel.

6. Implementation

6.1. Processing a PAR in the Control String

The algorithm used to process PAR in the control string is the following one. It is given in pseudo C language notation.

```
function processControlString (*c1 : controlString) -> rule ;
        controlString *c2;
begin
        while (c1 <> NULL) do
        begin
                switch (c1->type) /* on the type of control structure being processes */
                ...
                        case PAR:
                                /* process each par parameter */
                                /* the first parameter is handled by the parent */
                                /* each remaining parameter is handled by a child process */
                                c2 = c1->parameterList;
                                c2 = c2->next;
                                i=0;
                                while (c2 <> NULL) do
                                begin
                                        i++;
                                        if (c2 has no pertinent rules) then
                                                begin
                                                        idChild[i] = 0;
                                                        c2 = c2->next;
                                                end
                                        else
                                                begin
                                                        dumpRelations (c2);
                                                        pipe (pipes[i]);
                                                        pid = fork();
                                                        if (pid is the parent) then
                                                                begin
                                                                        idChild[i] = pid;
                                                                        c2 = c2->next;
                                                                end
                                                        else begin

                                                                /* prepare the child */
                                                                serverId = i;
                                                                fp = fdopen (pipe in write mode);

                                                                /* eliminate all other PAR parameters*/
                                                                c2->next = NULL;

                                                                /* set DBMS communication link */
                                                                /* for child */
                                                                fdr = fdw = fdrw[serverId];

                                                                loadRelations();
                                                                c1 = c2;
                                                        end
                                                end
                                        end
                                processControlString (c1->parameterList);
                        end

                        case RULE:
                                if (the rule pointed to by c1
                                        ISIN the list of pertinent rules) then
                                        return (the rule pointed to by c1);
                        end;
                end
                c1 = c1->next;
        end
        return(NULL); /* no pertinent rule can be found in this parameter list */
end
```

The control string is the support for parallelism in the language. The control string is mapped to an in-memory data structure which is used by the inference engine. While processing the control string, if the inference engine encounters a Par control structure, the engine tries to fork child processes to run rules in parallel. To do this, it goes through the following steps. First, it determines, for a set of rules, if there are pertinent rules in the set. If there are no pertinent rules, it marks the fact that no

child process has been created to run the rules. Otherwise, it scans the rules in a set to determine which relations are manipulated by the rules. Each non-empty relation that has not been dumped is unloaded into the common load/dump area. Then, a pipe is created to communicate between the parent and the child and the fork system call is issued. The child sets a number of variables to identify and to communicate with the DBMS on which it will run its set of rules. It attempts to load the relations which have been dumped by the parent process. It will then run independently from the parent until it has fired all the rules in its set. At this point, it will synchronize with the parent.

Version numbers are used to determine if a relation has been dumped. Each time a relation is modified by the RHS of a rule, its version number is incremented. When scanning relations to unload them into the common area, the version number is compared to the dump number of the relation. If it is less, the relation is unloaded and the dump number is set to the version number. The unload is realized by the SQL-like Copy command.

The child process scans all relations which are referenced in its set of relations and loads those relations which are not empty and have not yet been loaded. Again, version numbers are used to determine if the DBMS process associated to the child already has a relation in it memory. If the version number of the child process is less than the parent's, the relation is loaded.

7. Performance Analysis

7.1. The Environment

The objective of this section is to measure the performances of the execution of a RDL//C program. The Sabrina DBMS [Gardarin89] we used, implements all the standard features of commercial relational DBMS. The system architecture is a network of Sun SPARK workstations, each one having disk capabilities to store the database. This environment is somewhat different from bus-based backend database processing. Particularly, workstations are not designed for stand-alone processing (as dedicated database servers). Rather, they offer resource sharing to obtain load balancing through a local area network. However, such an environment can provide a larger memory space and faster computation in many cases. Moreover, a network of workstations constitutes a very common environment which has the advantages of parallel processing but without the specialized or dedicated hardware.

7.2. The Application

The application used to measure performance is based on a typical transitive closure operation to solve the ancestor problem. A sequential version of the rule module which calculates ancestors has been given in Section 2. We give a parallel program for the same problem. Parallelization of transitive closure has been widely studied [Valduriez88, Agrawal88, Cheiney90]. Within proposed algorithms, parallelism is included in the operator itself. In our approach, we express the

parallelism within the rule language by adding localization predicate to rules [Wolson90] and using the PAR structure within the control string.

The relation Parent is duplicated on n nodes. The processing task is divided into n parts: a first station computes the ancestors of desc value whose (value MOD2) = 0 ; a second station computes the ancestors of desc value whose (value MOD2) = 1; and so on, for n workstations. In the case of total replication of the relation Parent, the ancestors problem can be partitioned among n workstations without intermediate transfer. Only the building of the complete Ancestor relation implies transfers.

We give the RDL//C program for two nodes :

```
MODULE ancestor

CE[illegible] ;
        [illegible] and servB are two machines)
BASE
        Parent (asc int, desc int) ;

DEDUCED
        AncA LIKE Parent ;
        AncB LIKE Parent ;
        Ancestor LIKE Parent ;

RULES

r1A IS                                          r1B IS
        IF Parent (x) (x.desc MOD 2 = 0)                IF Parent (x) (x.desc MOD 2 = 1)
        THEN +AncA (x) ;                                THEN +AncB(x) ;

r2A IS                                          r2B IS
        IF Parent (x) AncA (y) (x.desc = y.asc)         IF Parent (x) AncB (y) (x.desc = y.asc)
        THEN +AncA (asc = x.asc, desc = y.desc) ;       THEN +AncB (asc = x.asc, desc = y.desc) ;


r3 IS
        IF AncA(x) THEN +Ancestor(x) ;

r4 IS
        IF AncB (x) THEN +Ancestor(x) ;

CONTROL

        PAR (SEQ (r1A, BLOCK (r2A)), SEQ (r1B, BLOCK (r2B))) ;

END MODULE
```

We will compare the performances of the system varying the load and the number of processors available. The load will vary according to the number of tuples in the Parent relation. Measurements are done with different loads. These loads are generated as a tree of ancestors, which has h levels (Figure 7.1). h is also the number of firings of the recursive rule (i.e. the depth of the induced join loop in our implementation). There are no indexes on the relations. The number of tuples vary in each set by having the number of descendants at each node vary by one. In the first set of tuples each node has three descendants; in the second set, each node has four; and in the last set, each node has seven descendants. With h=4, we obtain the five following loads: 363 tuples, 1364 tuples, 3905 tuples, 9330 tuples, 19607 tuples. The number of processors varies from one to five.

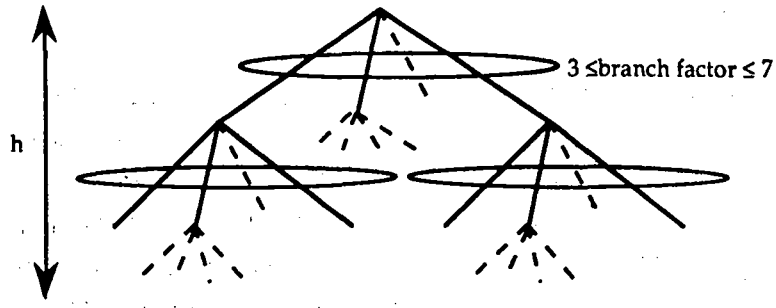

Figure 7.1: *Tuples generation tree*

7.3. Performance Results

To obtain performance results, we use some system time routines provided by the UNIX System. Eight time measurements are investigated, that are :

(1) Sql_connection. The time measured includes starting up a DBMS process on each remote station. This time is *a priori* proportional to the number of stations ;

(2) Validation of the schema of base relations at run-time ;

(3) Dump/Load/Wait. Using the NFS, passing relations consists in loading and dumping relations in files ; running parallel processes are synchronized by waiting for each one to have finished running the assigned set of rules. This time measures transfer and synchronization tasks (i.e. the parallelism overhead) ;

(4) Left Hand Side (LHS) of rule processing. This time measures select/join processing time ;

(5) Sql_get_schema. The schema is obtained at run-time ;

(6) Validation of the schema of the deduced relations at the run-time ;

(7) Right Hand Side (RHS) of rule processing. This measures the computation time of the union ;

(8) Sql_disconnect measures the time to disconnect the applications from the DBMS.

The first results have shown that only three of these measurements are significant within the overall time : Dump/Load/Wait, LHS and RHS rule processing. The sum of the other five components represents always less than 3% of the overall time (except for a few tuple -363 tuples- and a single processor where the overhead time -connection, deconnection, validation of schema- reaches to 5%). For this reason, the main point of discussion will be the three main time-consuming actions.

7.3.1. Sequential Time

The first measurements are the times with a single workstation. The aim was to reveal the behavior of the rule program and the relative amount of execution time spent in each part of the program. Particularly, the time spent to evaluate the *condition part* of the rules (the LHS processing corresponds to a selection and a join) and the time spent to process the *action part* (the RHS processing corresponds to an union). Right and Left parts of rules compose the computation time. An external time, obtained as the sum of the connections, disconnections, validations of schema, is also calculated. Table 1 gives the measurements of the three main components of the elapsed time.

	363 tuples	1364 tuples	3905 tuples	9330 tuples	19607 tuples
connection, disconnection and schema control	3s	7s	8s	10s	12s
Left Hand Side processing (Select/Join)	12s	339s	923s	2309s	4490s
Right Hand Side processing (Union)	41s	451s	1553s	4601s	9571s

Table 1: *Time spent in part for sequential processing*

Table 1 shows the importance of union processing time. Moreover, these measurements make clear that the external time is very limited and can be neglected when processing large sets of tuples (from 5% of the overall time for a load of 323 tuples to less than 1% for a load of 19,607 tuples). The overhead time due to rule program connection and controls, is very small ; the computational power is almost completely used to process the rules.

7.3.2. Parallel Time

Neglecting the connection time, the components of the total time are the join computation, the union computation and the parallelism overhead including the dumps, the loads and the waiting. We measured the overall and the partial times for five sets of basic tuples : the first one (363 tuples) produced 1641 ancestors, the second one (1,364 tuples) produced 6,372 ancestors, the third one (3,905 tuples) produced 18,555 ancestors, the fourth one (9,330) produced 44,790 ancestors and the fifth one (19,607) produced 94,773 ancestors. The number of workstations ranged from one to five.

Figure 7.2 presents a summary of these results. The times spent within the union and the select/join processing are represented by the grey areas. The black areas concern the time used to connect and disconnect the application and the DBMS processes. They can be neglected for a large amount of tuples. Last, the white areas summarize the overhead introduced by the parallelism, i.e. the time spent loading and dumping relations and files, and synchronizing. These times are cumulated to illustrate the overall response time.

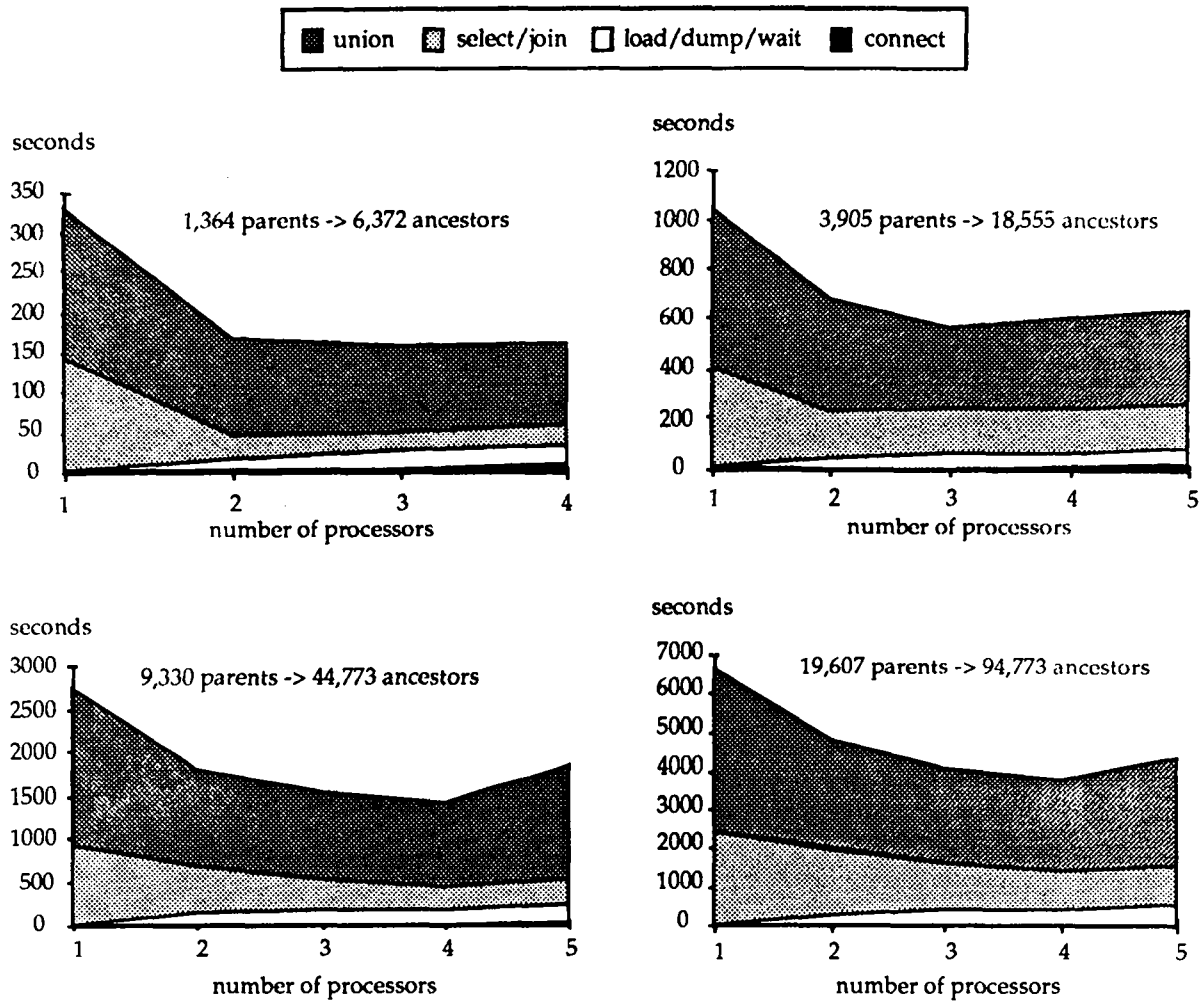

Figure 7.2 : *Evolution of overall and partial times in parallel processing*

7.3.3. Discussion

Figure 7.2 shows the benefits and limitations of parallelism in our experiment. Two points have to be discussed.

The first one concerns the application. The curves show that the join and the union processing times are not divided by the number of processors. The partition of the processing does not divide the work into the number of processor. The complete Parent relation has indeed to be read at each iteration *on each server*. The corresponding amount of time increases with the number of processor. The benefit is concentrated in the parallel generation of new tuples, without redundant production, which allows the response time to decrease with an increase in the number of processors. For 4 processors or more, the overhead of the multiple reads of Parent weighs over the benefit due to the division of the new tuples generation. However, this experiment shows that the use of three nodes can divide the response time by a factor of two. Note that the grain of parallelism (the rule) is determinant to partition the computation task. To obtain a more precise parallelism, the programmer has to write more rules (e.g. to specify a more sophisticated algorithm to solve the ancestor problem, including

the partition of Parent). In our experiment, we have chosen a simple program, but other example may yields better parallelism.

The second remark concerns the overhead induced by the parallelism. This overhead is represented on Figure 7.2 by the white area. Our experiments show that its augmentation, as a function of the number of processors, is limited. The parallelism is thus a good solution in our case. However, it is necessary to note that the transfers in our program are reduced to computation of the final result. In the case of more sophisticated algorithms, transfers are necessary during the ancestors generation ; the overhead due to these transfers would increase, especially in our configuration (a network of workstations) which is not an ideal hardware to reach high performances of parallelism.

Efficiency of the parallelism with a grain of a rule, is largely influenced by the user's choices in his RDL//C program. There is a traditional tradeoff between a good partition of computation tasks and a limitation of the transfers.

We have also tried to evaluate the influence of the number of tuples on the benefit provided by using multiple workstations. Figure 7.3 draws the overall time as function of the number of basic tuples. The maximum benefit is achieved for three or four stations for any number of tuples.

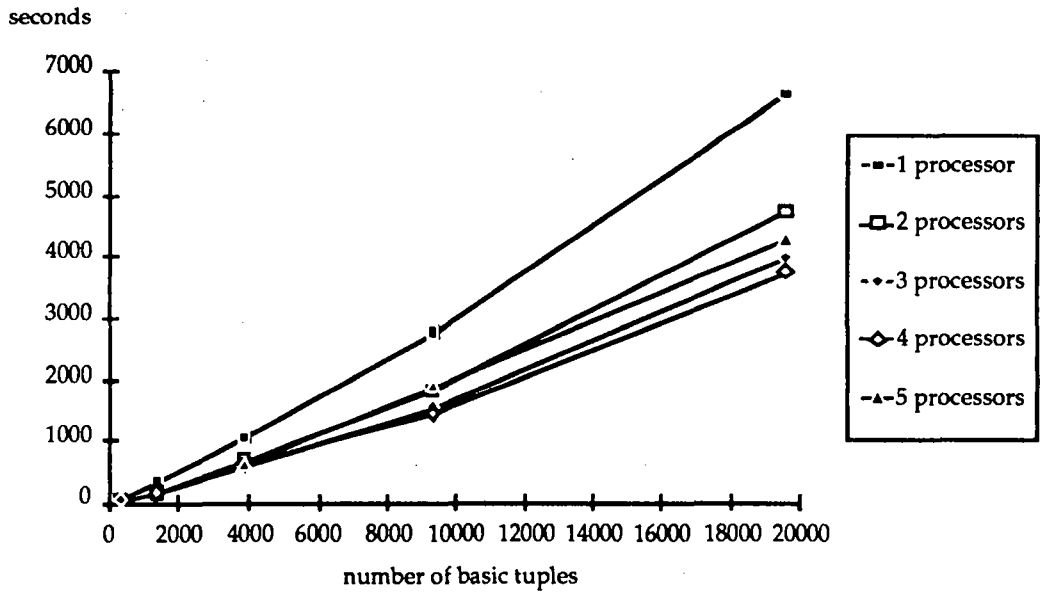

Figure 7.3: *Overall time as a function of the number of basic tuples*

During the measurements, the database was duplicated on each station's disk. We have also experimented the storage of all the databases on a single disk; because a large main memory was available within each station, the results were rather identical.

8. Conclusion and Future Work

This paper presented a database rule language compiler which supports parallelism among rules. The type of parallelism supported by the language is user-specified, as opposed to user-transparent

parallelism. The user partitions his application into independent tasks, which are scheduled to run in parallel. The compiler checks that the sets of rules to be run in parallel are independent and that the number of sets does not exceed the number of servers available.

The run-time framework for the parallel execution of database rule based applications is a set of UNIX workstations connected on a network. While the application process resides on one workstation, it drives DBMS processes which are on different workstations, each with its own database. Parallelism is achieved by having the application process fork into N processes, each one assigned a set of rules to be processed on one DBMS. Relations are transferred among the different DBMS by dumping relations onto files and loading relations from files using a common file sub-system managed by the network.

The main contribution of the paper are (i) the definition of a sub-language of control to specify a parallel execution of a rule program and (ii) the support of parallelism without specialized hardware and with standard relational technology. A prototype of RDL//C is operational over a set of UNIX workstations. Future works will study an automatic generation of the control structure which specifies the parallelism and the use of the RDL//C system to prototype parallel database algorithms that have been developed.

9. References

[Agrawal88] R. AGRAWAL, HV. JAGADISH: "Multiprocessor Transitive Closure Algorithms", In *Proceedings International Symposium on Databases in Parallel and Distributed Systems*, Austin, Texas, USA, December 1988.

[Almasi89] G. ALMASI, A. GOTTLIEB: *"Highly Parallel Computing"*, Redwood City, CA The Benjamin/Cummings Publishing Company, Inc., 1989

[Brownston85] L. BRONSTON, R. FARRELL, E. KANT, M. MARTIN: "Programming Expert Systems in OPS5 : An Introduction to Rule-Based Programming", Book, Addison-Wesley, 1985.

[Cheiney90] J.-P. CHEINEY, C. de MAINDREVILLE: "A Parallel Strategy for Transitive Closure Using Double Hash-based Clustering" In *Proceedings 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[Ganguly90] S. GANGULY, A. SILBERSCHATZ, S. TSUR: "A Framework for the Parallel Processing of Datalog Queries", In *Proceedings of ACM SIGMOD 1990 International Conference on Management of Data*, Atlantic City NJ, USA, June 1990.

[Gardarin89] G. GARDARIN, J.-P. CHEINEY, G. KIERNAN, D. PASTRE, H. STORA: "Managing Complex Objects in an Extendible Relational DBMS", In *Proceedings 15th International Conference on Very Large Databases*, Amsterdam, The Nederlands, August 1989.

[Genrich86] H. J. GENRICH: "Predicate / Transition Nets " , In *Advances in Petri Nets' 86*. Springer Verlag, 1987.

[Gupta89] A. GUPTA. et al.: " High-Speed Implementations of Rule-Based Systems", In *ACM Transactions on Computer Systems*, Vol 7, N° 2, May 1989.

[Kiernan90a] G. KIERNAN, C. de MAINDREVILLE , E. SIMON: "Making Deductive Database a Practical Technology: a step forward", In *Proceedings of SIGMOD 1990 International Conference on Management of Data*, Atlantic City NJ, USA, June 1990.

[Kiernan90b] G. KIERNAN, C. de MAINDREVILLE: "The RDL/C Language Reference Manual", INRIA Technical Report N° 123, October 1990.

[Kiernan91] G. KIERNAN, C. de MAINDREVILLE: "Compiling a Rule Database Program into a C-SQL Application", to appear in *Proceedings IEEE 7th International Conference on Data Engineering*, Kyoto, Japan, April 1991.

[Maindreville88] C. de MAINDREVILLE , E. SIMON: "Modelling Non-deterministic Queries and Updates in Deductive Databases", In *Proceedings 14th International Conference on Very Large Data Bases*, Los Angeles, USA, 1988.

[Srivastava89] J. SRIVASTAVA et al.: "Parallelism in Database Production Systems", In *Proceedings IEEE 6th International Conference on Data Engineering*, Los Angeles, USA, February 1990.

[Unix 84] Unix System V. User Reference Manual

[Valduriez88] P. VALDURIEZ, S. KHOSAFIAN: "Parallel Evaluation of the Transitive Closure of a Database Relation", In *International Journal of Parallel Programming*, Vol 17, N°1, February 1988.

[Wolfson90] O. WOLSON, A. OZERI: "A New Paradigm for Parallel and Distributed Rule Processing", In *Proceedings of ACM SIGMOD 1990 International Conference on Management of Data*, Atlantic City NJ, USA, June 1990.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399